EJ187645716US

Docket No. AUS990800US1

MANAGING AND EXTENDING ATTRIBUTE VALUES FOR PUBLIC KEY CRYPTOGRAPHY STANDARDS

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates generally to an improved data processing system and in particular to a method and an apparatus for implementing Public Key Cryptography Standards (PKCS). Still more particularly, the present invention provides a method for manipulating PKCS-attributes and user-defined attributes in a PKCS compliant system.

2. Description of the Related Art

Public Key Cryptography Standard (PKCS) is a set of documents published by RSA Laboratories and serves to define data types and algorithms used in public-key cryptography. Public-key cryptography is a technology in which encryption and decryption involve the use of a *public key* and a *private key*, and either can encrypt and/or decrypt data. A user gives his or her public key to other users, keeping the private key to himself or herself. Data encrypted with a public key can be decrypted only with the corresponding private key, and vice versa.

The PKCS set of standards has been developed to assure that software using cryptography at two different sites could work together even when the software is developed by different vendors for a variety of purposes. In particular, standards are being developed to allow agreement on digital signatures, digital enveloping,

digital certification, and key agreement. However, interoperability requires strict adherence to communicable formats, and PKCS provides a basis for interoperable standards in heterogeneous environments.

The present set of PKCS standards includes:

PKCS #1: RSA Encryption Standard;

PKCS #3: Diffie-Hellman Key Agreement Standard;

PKCS #5: Password-Based Encryption Standard;

PKCS #6: Extended-Certificate Syntax Standard;

PKCS #7: Cryptographic Message Syntax Standard;

PKCS #8: Private-Key Information Syntax Standard;

PKCS #9: Selected Attribute Types;

PKCS #10: Certification Request Syntax Standard;

PKCS #11: Cryptographic Token Interface Standard;

PKCS #12: Personal Information Exchange Syntax Standard;

PKCS #13: Elliptic Curve Cryptography Standard; and

PKCS #15: Cryptographic Token Information Format Standard.

Two independent levels of abstraction have been provided by these standards. The first level is message syntax, and the second level is specific encryption algorithms. The intention has been that message syntax and specific algorithms should be orthogonal. In other words, a standard for the syntax of digitally signed messages should be able to work with any public-key algorithm, not just RSA, the public-key algorithm invented by Rivest, Shamir, and Adleman involving exponentiation modulo the product of two large prime numbers; and a standard for RSA should be applicable to many different message syntax standards.

PKCS provides definitions of data objects that may be created, sent, and received between parties to a communication, while other standards are used to define the encoding syntax of the data streams containing these types of data objects. Abstract Syntax Notation One, abbreviated ASN.1, is a notation for describing abstract types and values. The Basic Encoding Rules (BER) for ASN.1 give one or more ways to represent any ASN.1 value as an octet string. The Distinguished Encoding Rules (DER) for ASN.1 are a subset of BER, and give exactly one way to represent any ASN.1 value as an octet string. DER is intended for applications in which a unique octet string encoding is needed, as is the case when a digital signature is computed on an ASN.1 value. ASN.1 and DER encoding are general purpose methods that can be applied to many domains in addition to PKCS.

One of the PKCS standard documents, PKCS #9, defines a set of attributes that can be used in other PKCS standards. PKCS #9 defines selected attribute types for use with various types of other data objects within other PKCS standards, such as PKCS #6 extended certificates and PKCS #7 cryptographic messages. For example, PKCS #7 defines the syntax for several cryptographically protected messages, including encrypted messages and messages with digital signatures. PKCS #7 also allows arbitrary attributes, such as signing time, to be authenticated along with the content of a message. Originally an outgrowth of Internet Privacy-Enhanced Mail, PKCS #7 has become the basis for the widely implemented Secure/Multipurpose Internet Mail Extensions (S/MIME) secure electronic mail specification, an Internet e-mail security standard that employs public key

encryption. PKCS #7 has become a basis for message security in systems as diverse as the Secure Electronic Transaction (SET) specification for bank systems.

Since PKCS #9 provides attributes to support other components in the PKCS set of standards, it is important that any software that processes PKCS #9 attributes handle these attributes in a robust manner. For example, a software application could be written such that it has knowledge of attributes defined as part of the PKCS #9 standard, and the application could treat unknown attributes as "undefined" attributes. However, given the fact that this set of standards continues to evolve, it is important for software that handles PKCS #9 attributes can handle "extended" attribute or user-defined attributes. Beyond the attributes that are defined as part of the standard, a software developer may desire to process a user-defined set of attributes along with the standard list of attributes.

Therefore, it would be advantageous to have an architecture and a method for manipulating PKCS attributes that allows a data processing system to be extended to accommodate additional attributes.

SUMMARY OF THE INVENTION

A method and system for processing PKCS-attributes and user-defined attributes in heterogeneous environments is provided. Attributes are registered with a PKCS9 gateway class, and the attributes include user-defined attributes and PKCS-standard defined attributes. Each of the registered attributes is associatively stored with an identifier. An object-oriented method in the PKCS9 gateway class may be called with a parameter containing an object identifier for an attribute. An attribute mapping data structure is searched using the object identifier in the received parameter, and in response to finding a matching object identifier, a class identifier that has been associatively stored with the matching object identifier is retrieved from the attribute mapping data structure. A method in the class identified by the class identifier is then called. The called method may include an operation for construction, attribute conversion to and from DER-encoding, attribute differentiation, and attribute value extraction. A class hierarchy of attribute types is based on an abstract class for all attribute objects with a subclass for undefined attributes and a subclass for defined attributes. The subclass for defined attributes is further decomposed into a subclass for each PKCS-defined attribute and a subclass for each user-defined attribute.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself, however, as well as a preferred mode of use, further objectives and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

Figure 1 is a exemplary representation of a distributed data processing system in which the present invention may be implemented;

Figure 2 is an exemplary block diagram of a data processing system that may be implemented as a server;

Figure 3 is a diagram showing the Abstract Syntax Notation for the X.500 Attribute type;

Figure 4 is a diagram showing two examples of PKCS #9 attributes, each with sample values;

Figure 5 shows an example of DER-encoding in hexadecimal format of the two PKCS #9 objects shown in **Figure 4**;

Figure 6 is a class hierarchy diagram that shows an exemplary set of classes for use in the software methodology of the present invention for implementing PKCS-attribute-related operations in an extensible manner;

Figure 7A is a flowchart outlining an exemplary operation for requesting an attribute object via the PKCS9 class in accordance with an embodiment of the present invention;

Figure 7B is a diagram showing an example format of a user-defined attribute configuration file that maps attribute object identifiers to the classes that implement the attributes;

Figure 7C is a table showing an example of mappings from object identifiers to the classes that implement the attributes; and

Figure 8 is a flowchart outlining an exemplary operation of the getAttribute() method for the PKCS9 class in accordance with a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention provides a software methodology for managing PKCS #9 attributes that may be associated with a variety of data objects. These data objects are formatted according to other standards in the PKCS set of standards. The data objects and their associated attributes may be embedded within messages that are sent within a distributed data processing system between parties that desire certain types of security features provided by the PKCS standards.

With reference now to the figures, and in particular with reference to **Figure 1**, a pictorial representation of a distributed data processing system is depicted in which the present invention may be implemented.

Distributed data processing system **100** is a network of computers. Distributed data processing system **100** contains network **102**, which is the medium used to provide communications links between various devices and computers connected within distributed data processing system **100**. Network **102** may include permanent connections, such as wire or fiber optic cables, or temporary connections made through telephone connections.

In the depicted example, servers **104**, **114**, **116** and **118** are connected to network **102**. Storage units **106** and **122** are also connected to network **102**, providing backup support for any or all of servers **104**, **114**, **116** and **118**. In addition, clients **108**, **110** and **112** are also connected to network **102**. These three clients may be, for example,

personal computers or network computers. Distributed data processing system **100** may include additional servers, clients, and other devices not shown.

In the depicted example, servers **104**, **114**, **116** and **118** provide storage for data from clients **108, 110** and **112**. These four servers also provide data, such as boot files, operating system images, and applications to clients **108, 110** and **112**. Clients **108**, **110** and **112** are clients to one or all of servers **104**, **114**, **116** and **118**. In the depicted example, distributed data processing system **100** may be the Internet, with network **102** representing a worldwide collection of networks that use the TCP/IP suite of protocols to communicate with one another. At the heart of the Internet is a backbone of high-speed data communication lines between major nodes or host computers consisting of thousands of commercial, government, education, and other computer systems that route data and messages. Of course, distributed data processing system **100** also may be implemented as a number of different types of networks, such as, for example, an intranet or a local area network.

Figure 1 is intended as an example and not as an architectural limitation for the processes of the present invention.

The present invention may be embedded within a software application that provides some type of secure messages or cryptographic features to a user for a variety of purposes. For example, an application that employs the software methodology of the present invention may execute on any of the clients or servers shown in **Figure 1**. The application may use the present invention to create a data object representing an instance of a

PKCS #9 attribute, which is then associated with another PKCS-related data object and embedded within a PKCS #7 cryptographic message. The cryptographic message may then be sent between clients and/or servers in network **102**.

With reference now to **Figure 2**, a block diagram of an exemplary data processing system which may be implemented as a server or client, such as server **104** or client **108** in **Figure 1**. Data processing system **200** may be a symmetric multiprocessor (SMP) system including a plurality of processors **202** and **204** connected to system bus **206**. Alternatively, a single processor system may be employed. Also connected to system bus **206** is memory controller/cache **208**, which provides an interface to local memory **209**. I/O bus bridge **210** is connected to system bus **206** and provides an interface to I/O bus **212**. Memory controller/cache **208** and I/O bus bridge **210** may be integrated as depicted.

Peripheral component interconnect (PCI) bus bridge **214** connected to I/O bus **212** provides an interface to PCI local bus **216**. A number of modems **218-220** may be connected to PCI bus **216**. Typical PCI bus implementations will support four PCI expansion slots or add-in connectors. Communications links to network computers **108-112** in **Figure 1** may be provided through modem **218** and network adapter **220** connected to PCI local bus **216** through add-in boards.

Additional PCI bus bridges **222** and **224** provide interfaces for additional PCI buses **226** and **228**, from which additional modems or network adapters may be supported. In this manner, server **200** allows connections

to multiple network computers. A memory-mapped graphics adapter **230** and hard disk **232** may also be connected to I/O bus **212** as depicted, either directly or indirectly.

Those of ordinary skill in the art will appreciate that the hardware depicted in **Figure 2** may vary. For example, other peripheral devices, such as optical disk drives and the like, also may be used in addition to or in place of the hardware depicted. The depicted example is not meant to imply architectural limitations with respect to the present invention.

The data processing system depicted in **Figure 2** may be, for example, an IBM RISC/System 6000, a product of International Business Machines Corporation in Armonk, New York, running the Advanced Interactive Executive (AIX) operating system.

Before describing the software methodology of the present invention for processing PKCS #9 attributes, **Figures 3-5** provide some background information about the syntax and encoding of attributes that are used in the PKCS set of standards.

With reference now to **Figure 3,** a diagram shows the Abstract Syntax Notation (ASN.1) for the X.500 Attribute type. An attribute is defined in the X.500 standard to be essentially a coupling of a unique object identifier ~~(ID)~~ (OID) with some value. An attribute value can be composed of one or more items, and the data type of the items themselves depends upon the type of the attribute.

As stated, PKCS #9 defines a specific set of attributes. Currently, in the version 1.1 standard, there are nine such attributes, described in more detail below in **Figure 6**. The definition of each includes its

object identifier, the data type of its value, and whether its value is single-valued or multi-valued.

With reference now to **Figure 4,** a diagram shows two examples of PKCS #9 attributes, each with sample values. EmailAddress attribute **400** is an example of a multi-valued attribute. The object identifier **402** has two value items, **404** and **406**. These values have ASN.1 data type IA5String, where IA5String is a sequence of ASCII characters. ContentType attribute is a single-valued attribute **408**. The object identifier **410** has a single value **412** that is also an object identifier.

ASN.1 notation defines the data types, and any software application can implement data type definitions in any suitable way: a C struct, a Java object, and a Pascal record are possible implementations. These disparate objects must be encodable in some standard way to ensure interoperability among the software implementing the defined objects. One way to accomplish this uniform encoding is through the Distinguished Encoding Rules (DER), a method to transform an ASN.1-defined object into a byte array in standard format. In this way, a software entity on one system may implement an ASN.1 object in one fashion, encode it via DER, and transmit it to another software entity, which can then parse the DER byte array into its own representation of the same ASN.1 object originally encoded.

With reference now to **Figure 5**, a figure shows DER-encoding in hexadecimal format for the two PKCS #9 objects depicted in **Figure 4**. In particular, encoding **502** is for EmailAddress and encoding **504** is for

ContentType. The important feature of these encodings is that they are "distinguished" or unique.

As mentioned previously, an application may use the present invention to create a data object representing an instance of a PKCS #9 attribute, which may then be associated with another PKCS-related data object and embedded within a message sent across a network. There are basically four operations that must be implemented within a robust application to fully support PKCS #9 attributes:

(1) An application should be able to build an attribute from its constituent parts, its object identifier, and its value.

(2) An application should be able to convert an attribute object into a DER-encoded byte stream for transmission to other systems.

(3) An application should be able to convert a DER-encoded byte stream into an attribute object.

(4) An application should be able to distinguish one attribute from another and extract an attribute's value information.

The PKCS #9 architecture outlined in this invention supports these PKCS attribute operations, such as construction, attribute conversion to and from DER-encoding, attribute differentiation, and attribute value extraction. By implementing the software methodology of the present invention in an object-oriented programming language, such as Java, some of these operations may be facilitated by having very specific attribute classes while other operations are facilitated by more general classes. The class hierarchy described below combines elements of specific and general

classes to implement PKCS-attribute-related operations in a manner which is extensible to user-defined attributes. "User-defined attributes" may include attributes that have specific user purposes in specific applications, i.e. attributes which are not contemplated as eventually becoming standard-defined attributes or are not promulgated as standard-defined attributes.

In addition, a "user-defined attribute" may also include attributes that were not part of the PKCS #9 standard when a software application implementing the present invention was deployed. At some later time, when an attribute that was previously recognized as a user-defined attribute then becomes part of the PKCS #9 standard, the application may be updated so that the application recognizes the PKCS #9 standard-defined attribute and so that the attribute is no longer recognized as a user-defined attribute. In this manner, the present invention allows an application to be extended to user-created attributes and subsequent, standard-defined attributes.

With reference now to **Figure 6**, a class hierarchy diagram shows an exemplary set of classes for use in the software methodology of the present invention for implementing PKCS-attribute-related operations in an extensible manner. PKCSDerObject **600** is an abstract class representing any object that can into encoded into or decoded from a DER byte stream. All subclasses of PKCSDerObject must implement encode and decode methods.

DefinedAttribute **602** is an abstract class representing any attribute that is "known" to the system; that is, those attributes that have individual class implementations and that are registered with the PKCS9

class. The registration process is discussed below with respect to the PKCS9 gateway class **610**. The subclasses of DefinedAttribute **602** must implement a number of methods to return attribute-specific information such as the attribute's common name and whether the attribute is single-valued or multi-valued.

PKCS #9 standard-defined-attribute Classes **604** are a set of classes representing the attributes defined in PKCS #9. As of version 1.1 of the PKCS #9 standard, there are nine defined attributes: ChallengePassword, EmailAddress, SigningTime, ContentType, ExtendedCertificateAttributes, UnstructuredName, Countersignature, MessageDigest, and UnstructuredAddress. Additional classes can be added to this set as further versions of the PKCS #9 standard are developed and approved. All of these classes implement the abstract methods in their parent class, DefinedAttribute **602**.

User-defined-attribute Classes **606** are a set of classes representing ASN.1 Attribute types that are not defined in the PKCS #9 standard (and therefore not represented by a class in PKCS #9 standard-defined-attribute classes **604**) but that are known to the system and registered with the PKCS9 gateway class **610**. Like the standard PKCS #9 standard-defined-attribute classes **604**, user-defined-attribute classes must implement the abstract methods in DefinedAttribute.

UndefinedAttribute **608** represents any attribute that is unknown to the system. That is, there is not a specific class that implements the attribute, so the PKCS9 class **610** cannot map the attribute object identifier to an implementing class. The

UndefinedAttribute class **608** allows the decoding of arbitrary attributes that may be undefined on the system. The UndefinedAttribute class **608** includes an ObjectIdentifier data member which is the attribute OID and a byte array data member which is the DER-encoding of the attribute value.

PKCS9 gateway class **610** serves as the gateway to all the attributes on the system, both known and unknown. It has a variety of getAttribute() methods that allow easy instantiation of an attribute by OID and value, string name and value, or DER-encoding. PKCS9 gateway class **610** will interpret the input and return an instance of the proper DefinedAttribute or an UndefinedAttribute.

All attributes defined in the PKCS #9 standard (and therefore represented by a class in PKCS #9 standard-defined-attribute classes **604**) are statically registered within the PKCS9 gateway class **610**. That is, the PKCS9 gateway class **610** has a mapping of each attribute's unique OID to the class that implements it. To register user-defined attributes, the user must first implement the attribute as a subclass of DefinedAttribute since the PKCS9 gateway class **610** uses the abstract methods defined in DefinedAttribute and then enters the attribute OID and its implementing class in a configuration file.

Figure 6 shows two user defined attributes: "NetscapeComment", an attribute used in many applications yet not one of the attributes defined in the PKCS standard, and "SomeAttribute", an exemplary name chosen to suggest a user-defined attribute for a particular application. When the PKCS9 gateway class **610** is first loaded, it will read a configuration file and internally

store the mapping information for OIDs of user-defined attributes. Thereafter, when a getAttribute() method is called and the PKCS9 gateway class **610** detects a user-defined OID in the input arguments, it will be able to access the proper class for attribute instantiation. This registration process and the basic means of obtaining an attribute via the PKCS9 gateway class **610** is discussed in more detail below with respect to **Figure 7**. The logic followed by the getAttribute() method is discussed in more detail further below with respect to **Figure 8**.

Note that a configuration file may be in a default location or may be in a location defined by a Java property defined in the java.security configuration file. If a configuration file is property-defined, the PKCS9 gateway class **610** must have permission to read the property. In either case, the PKCS9 gateway class **610** must have permission to read the configuration file itself.

Before discussing flowcharts for the algorithms associated with the present invention, it is useful to discuss the PKCS-attribute-related operations of construction, attribute conversion to and from DER-encoding, attribute differentiation, and attribute value extraction in more detail.

Constructing an attribute using the classes defined in the present invention can be accomplished in different ways, depending upon what information the user has at the time. If the type of the required attribute is known, then that attribute may be instantiated directly using a constructor that takes a meaningful data type. For example, if an UnstructuredName attribute is needed, that

attribute may be instantiated by calling the UnstructuredName constructor that takes a String argument (the name). There is no need to convert the name value to the DER-encoding of its ASN.1 data type. As another example, the ChallengePassword attribute may be instantiated by calling the ChallengePassword constructor that takes a char[] argument (the password). Note that security-sensitive data, such as passwords, should not be represented as String data types which are immutable.

If the user has an object identifier (OID) and some value, but does not know to which attribute class the OID corresponds, the PKCS9.getAttribute(ObjectIdentifier, Object) method associated with the PKCS9 class, such as the PKCS9 gateway class **610** shown in **Figure 6**, may be used to construct the attribute. The PKCS9 class will return an instance of the correctly defined attribute if the attribute OID is known to the system or an instance of UndefinedAttribute if the OID is unknown to the system.

If the user has only a DER-encoded byte stream, the PKCS9.getAttribute(byte[]) method may be used. This method will extract the attribute OID from the byte stream and then instantiate and return the proper attribute object.

Another operation is the conversion of an attribute object to a DER-encoded byte stream. Because all attributes are subclassed from PKCSDerObject, each must implement an encode() method. Converting an attribute object to its DER-encoding requires a simple call to the encode() method.

The reverse conversion of a DER-encoded byte stream to an attribute object is also possible. Because all

attribute classes are subclassed from PKCSDerObject, each attribute class must implement a decode() method. Explicitly creating an attribute object from a DER byte array requires using the attribute's default constructor and then calling that object's decode() method. Alternatively, the PKCS9 gateway class, such as PKCS9 gateway class **610** in shown in **Figure 6**, can be used to convert any DER-encoded attribute to an attribute object, and this relieves the user from having to know how to instantiate an attribute object explicitly by knowing the attributes type and calling the appropriate constructor.

Other useful operations are attribute differentiation and value extraction. An important usability feature of attributes is the ability to easily distinguish them from one another. Having a separate class implement each attribute allows the user to search for a particular attribute type using the "instanceof" operator. For example, X.509 extended certificates can include attributes. To find the signing time of an extended certificate, one could cycle through the attribute objects in the certificate object using the "instanceof SigningTime" comparator. There is no need to examine OIDs values within the attribute objects to find the correct one.

It is also preferable to be able to extract the attribute value in a form that is immediately meaningful, rather than in a form that must be converted before use. For example, the value of an EmailAddress attribute can be extracted directly into an array of Strings. This relieves the user from having to convert an internal representation of the value -- whether it be a DER byte

stream or a series of Objects, for example -- into Strings after value extraction.

With reference now to **Figure 7A**, a flowchart shows the request of an attribute object via the PKCS9 gateway class. The PKCS9 gateway class is loaded (step **702**). Upon loading, the static initializer block in this class is run. Static initialization includes reading in the data from the user-defined-attribute configuration file.

The user-defined-attribute configuration file is read (step **704**). The file is stored on disk either in a default location or in a location specified by a Java security property. If the latter, the PKCS9 gateway class must first read the Java property to find the file location, then access the file itself.

In the course of execution, a Java program may request an attribute instance by calling a PKCS9.getAttribute() method, passing in the appropriate arguments (step **706**). The logic followed by the PKCS9.getAttribute() method (step **708**) is described more fully in **Figure 8**. If there are more attributes to process (step **710**), then control is transferred to step **706**. Otherwise, the request of the Attribute object is completed.

With reference now to **Figure 7B**, a diagram shows an example format of a user-defined attribute configuration file that maps attribute object identifiers to the classes that implement the attributes. The PKCS9 gateway class reads mappings, e.g., at initialization time, in configuration file **712** and stores them internally in an appropriate data structure. In this example, object identifier **714** maps to class **716** and object identifier **718** maps to class **720**.

With reference now to **Figure 7C**, a table shows an example of mappings from object identifiers to the classes that implement the attributes. The first nine entries are PKCS #9 standard-defined-attribute classes where both the object identifier and class names are predefined. Since these values are fixed, they do not need to be stored in the configuration file. Rather, the information in the configuration file is added to the attribute mappings after the predefined values. As one of ordinary skill in the art will appreciate, the attribute mappings shown in **Figure 7C** could be implemented as a hash table, an association list, or similar data structure. This set of mappings will contain the object identifier and the class name for all registered attributes, including the PKCS #9 standard-defined-attribute classes and extended or user-defined attributes.

When a "get"-type method in the PKCS9 gateway class is invoked, the object identifier is extracted from the argument, the class name associated with the object identifier is determined from the set of mappings, and a constructor in the associated (i.e. mapped) class is called. The PKCS9 "get" method then returns an instance of a specific defined attribute, or an instance of an UndefinedAttribute if the object identifier does not have a class mapping.

With reference now to **Figure 8**, a flowchart shows the logic of the getAttribute() method for the PKCS9 gateway class. This figure is an expansion of step **708** in **Figure 7A.** Once a PKCS9.getAttribute() method is called, the attribute object identifier is extracted from the input arguments (step **800**). Depending on the

signature of the getAttribute() method, the OID may be in String form, in ObjectIdentifier form, or in a DER-encoded byte array. In each case, the PKCS9 gateway class is able to convert the OID to the form in which the object identifiers in the OID-class mappings for the PKCS9 gateway class are stored.

The PKCS9 class determines if the requested OID is one that has been registered, either statically (as an attribute class corresponding to the standard-defined PKCS #9 attributes) or at runtime through the user-defined attribute configuration file (step **802**).

If the PKCS gateway class has found a registered OID, then a constructor of the class mapped to that OID is called with the value argument passed to the PKCS9.getAttribute() method (step **804**). The specific attribute constructor will return an attribute instance to the PKCS9.getAttribute() method.

If the PKCS9 gateway class has not found a registered OID, then the constructor of the UndefinedAttribute class is called (step **806**). This constructor will return an UndefinedAttribute instance to the PKCS9.getAttribute() method. The PKCS9.getAttribute() method returns the attribute back to its caller (step **808**).

The programming language for the preferred embodiment of this invention is Java, but as one of ordinary skill in the art will appreciate, implementation could be in a wide variety of programming languages, such as C, Pascal, C++, and so forth. Implementation in an object-oriented language, such as C++, would be easier than in a language like C that is not object-oriented.

In summary, the architecture outlined in the present invention provides several usability features for PKCS attributes, including those that are not defined in the PKCS #9 standard:

(1) Distinct classes for each defined attribute allows for easy instantiation either directly or through the PKCS9 gateway class.

(2) Distinct classes for each defined attribute allows for easy differentiation between attributes. A specific attribute type can be found by using the "instanceof" operator with no need to examine internal data members.

(3) Distinct classes for each defined attribute allows attribute values to be extracted in forms meaningful to the user. For example, the value of a SigningTime attribute is known to be a Date object, and is directly retrievable and usable as such. There is no need to extract values generically as Objects or as byte arrays (DER-encodings) which then must be converted to usable objects.

(4) All attributes implement encode and decode methods for easy conversion to and from DER byte streams.

(5) Both defined and undefined attributes are supported.

(6) The architecture is extendible so that new attributes can be implemented and registered with the system.

It is important to note that, while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in

the form of a computer readable medium of instructions and a variety of forms, and that the present invention applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media include recordable-type media such a floppy disc, a hard disk drive, a RAM, and CD-ROMs and transmission-type media such as digital and analog communications links.

The description of the present invention has been presented for purposes of illustration and description but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. For example, this general architecture can be utilized in a variety of operating systems. It can also be implemented in different programming languages. It can also be extended to include new PKCS attributes as the Public Key Cryptography Standards evolve over time. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.